Ana Milanova’s Research Statement

My research is in the area of programming languages, compilers, and security. More specifically, I work in static program analysis, which reasons about program behavior before program execution. I apply program analysis techniques to improve software security and productivity, and data privacy.

1 Present and Past Research

I summarize, in reverse chronological order, the three major research thrusts I have worked on since my promotion to associate professor. I have led the projects working with diverse teams of graduate and undergraduate students and fostering collaboration with leading researchers in each area.

1.1 Compilation and Optimization for Multi-Party Computation

**Motivation** Multi-party computation (MPC) allows $N$ parties $p_1, \ldots, p_N$ to perform a computation on their private inputs securely. Informally, security means that the secure computation protocol computes the correct output (correctness) and it does not leak any information about the individual party inputs, other than what can be deduced from the output (privacy).

MPC theory dates back to the early 1980-ies. Long in the realm of theoretical cryptography, MPC has seen significant advances in programming technology in recent years. These advances bring MPC closer to practice and wider applicability — MPC technology has been employed in real-world scenarios such as auctions, biometric identification, and privacy-preserving machine learning. The goal is to bring the technology to a level where programmers can write secure and efficient programs without commanding extensive knowledge of cryptographic primitives.

**My Work** The problem, therefore, is to build high-level programming languages and optimizing compilers for MPC. My work focuses specifically on an intermediate language and what we call backend-independent optimizations, in a close analogy to machine-independent optimizations in the classical compiler. The following figure summarizes the key idea:

We emphasize the MPC-IR intermediate representation and optimization over MPC-IR. As in classical compilers, we envision different front ends (e.g., WYSTERIA, VIADUCT) compiling into MPC-IR. MPC-IR exposes the linear structure of MPC programs, which simplifies program analysis and optimizations such as protocol mixing, SIMD-vectorization (which takes advantage of amortization at the circuit level) and scheduling. In the same time, MPC-IR is sufficiently “high-level” and amenable to analyses and optimizations that take into account control and data flow in a specific program.

Protocol mixing is an MPC-specific optimization. In a nutshell, MPC programming supports different protocols of sharing and computation, and protocols have inherent differences. There are two basic approaches in MPC protocol design: (1) Yao’s garbled circuits and (2) the secret-sharing paradigm known as GMW; GMW gives rise to two techniques, known as Boolean and Arithmetic. Very informally, the Arithmetic protocol allows for efficient arithmetic operations (e.g., multiplication) but inefficient boolean and logical operations (e.g., comparison and multiplexing). Alternatively,
Yao allows for less efficient arithmetic operations but more efficient boolean ones. One can convert Arithmetic shares into Yao shares and vice versa, but typically at a non-trivial cost.

The problem of assigning protocols to MPC-IR statements, along with necessary (minimal) share conversions, can be viewed as a program analysis problem. One notable result of our project is an algorithm that computes an optimal protocol assignment in polynomial time for 2 protocols [5]. Our other notable result is a compiler from a Python-like source (we call IMP Source) into the two leading MPC backends, MOTION and MP-SPDZ; we formalize MPC-IR and build a SIMD-vectorization optimization that leads to significant performance improvement compared to iterative code [6]. Yet the problem of optimal protocol assignment for 3 or more protocols (in polynomial time) is still open, and we will continue to work on it. We are especially interested in integrating mixing (even heuristic mixing) with SIMD-vectorization and other optimizations.

**Publications** The work on optimal mixing appeared in CCS’19 (ref. [5]) and the work on compilation and SIMD-vectorization will appear in CCS’23 in November (ref. [6]).

### 1.2 Principled and Practical Static Analysis of Python

**Motivation** While Python is a widely used programming language, particularly in data science programming, principled static analysis for Python is lacking. Originally, we set out to extract hyperparameter constraints from machine-learning libraries, expecting to reuse existing results, only to discover that even classical analyses such as 3-address code translation and points-to analysis remain open problems. Many existing analyses are ad-hoc traversals of the Abstract Syntax Tree (AST); they do not elaborate on the exact statements and expressions they analyze or the traversal semantics. Analysis is largely ad-hoc and unsound.

**My work** Therefore we ask, can we define static analysis for Python in a (more) principled way? Is it possible to reason about what parts of a Python program are analyzed soundly and what parts are analyzed unsoundly? Such reasoning grounds the analysis and builds confidence in analysis results, particularly when (speaking very informally) results originate from sound parts of the program and remain “untainted” by flows from unsound parts.¹

Our work takes a step in this direction. We define PetPy, a minimal Python syntax where we break Python constructs into interpreted and uninterpreted ones. Interpreted constructs receive a sound and precise interpretation; these are attribute accesses, calls, assignments and other constructs that are ubiquitous in code and affect flow of references. In contrast, uninterpreted ones are treated to a common fall-through interpretation, which, in general, is neither sound nor precise; examples of uninterpreted constructs (in our current treatment) are while statements and list comprehensions. We then define (1) 3-address code translation and (2) weakest precondition inference as principled interpretations over PetPy. While our analysis remains unsound, we elaborate on the exact constructs and interpretation semantics [12, 11].

We also define a soundness analysis to reason about the inferred weakest precautions. The weakest precondition analysis is a standard backwards reasoning: it starts at an exception and propagates a formula \( Q \) backwards towards the beginning of the function. For each statement \( s \) it steps over, where \( s \) can be a call, a for-loop, an assignment, etc., the analysis reasons whether the locations modified by \( s \) intersect with the locations read by \( Q \). If the answer is yes, then \( Q \) becomes unsound.

**Publications** The work was published at ISSTA’22 (ref. [12]) and won an ACM SIGSOFT Distinguished Paper Award. Our extended version was just accepted to Wiley Software: Practice and Experience (SPE) (ref. [11]). In addition, our analysis (recall that our goal was to infer hyperparam-

¹We remark that “sound” in our treatment is in fact “soundy” in the spirit of the Soundness manifesto [7]. Soundness refers to static analyses that intentionally under-approximate certain language features because handling of those features over-approximately would render the analysis useless.
eter constraints encoded by exceptions in machine learning libraries) uncovered bugs in both code and documentation in popular machine learning libraries. We submitted issues and pull requests to sklearn, TensorFlow and numpy, which developers merged promptly.

1.3 Inference and Checking of Context-sensitive Pluggable Types

Motivation Static type systems improve software productivity and security by catching errors and security vulnerabilities before program execution. They typically require writing type annotations and the process can be burdensome to the programmer. Consequently, dynamically-typed languages like Python forgo static type systems and relegate essentially all type checking and error prevention to the runtime. Statically-typed languages like Java aim to lower the annotation burden as well — they check for many errors, but not all; for example, they do not check for unintended mutation or flow of sensitive values to untrusted components.

Pluggable types enhance a static type system. They can be plugged seamlessly into an existing program to catch bugs beyond the capabilities of the underlying static type system (if there is one). Notably, the CHECKER FRAMEWORK provides pluggable type systems that target different classes of Java bugs, including unintended mutation and SQL injection vulnerabilities. A downside is that a pluggable type system, by virtue of being a static type system, typically requires annotations by the programmer.

My work Our work focuses on type inference. Can we design pluggable type systems that require a minimal number of annotations, even no annotations at all? I led a project on a pluggable types framework that focused on context-sensitive reasoning (context-sensitive reasoning allows for verifying a larger number of programs as safe, as opposed to context-insensitive reasoning, which may flag safe programs as buggy). A type system designer specifies the essential components of a type system in the framework: the types, subtyping hierarchy and the context adaptation operation that enables context-sensitive reasoning. A programmer may add a small number of annotations in their program, or in many cases of type systems we designed, they add no annotations at all. The framework then automatically infers the rest of the annotations and type checks the program — it either verifies the program as safe (of the class of bugs that the type system checks for, e.g., unintended mutation), or flags errors (e.g., unintended mutation). The figure illustrates our framework:

The framework is general and gives rise to concrete instantiations (a subset is shown above). One is ReIm, a novel type system for reference immutability, and ReImInfer, the corresponding inference tool [4]. One can specify a small set of references as immutable and ReImInfer either flags
violations of their immutability, or verifies that they remain immutable across all program runs. The programmer can specify no annotations at all in which case ReImInfer infers a typing with a maximal number of immutable references. An extension of ReImInfer infers method purity, a fundamental analysis applicable in a variety of software engineering and compiler tasks. Other instantiations of the framework are the inference algorithms for classical Ownership types and Universe types, including the first polynomial-time algorithm for the inference of Universe types [1]. Another instantiation is DroidInfer, an information flow type system and a corresponding inference tool that finds information leaks in Android apps [2]. DroidInfer assumes annotated Android libraries that specify sources of sensitive data (e.g., device identifier, phone number) and untrusted sinks (e.g., an untrusted url). It requires no annotations by the programmer. DroidInfer flagged dozens of leaks in apps from the Google Play store (e.g., an advertising server stealing the device phone number).

In later work I generalized a class of context-sensitive pluggable type systems (including ReIm and DroidInfer) into a positive-negative type qualifier system and I reduced the inference problem to a CFL-reachability problem [8, 9]. The reduction enables reasoning about the semantics of the qualifier system and drives a correctness argument.

Publications This work appeared in OOPSLA’12 and ’20 (refs. [4] and [9]), ECOOP’12 and ’18 (refs. [1] and [8]), and ISSTA’15 (ref. [2]). An overview of our framework and a demo of the ReImInfer tool were published in the New Ideas and Tool Demo tracks of FSE’12, respectively (refs. [10] and [3]). This work has generated about 350 citations.

2 Future Research Directions

In the future, I will continue to work on compilers for secure computation and principled static analysis for Python. In addition to the research described earlier, which falls closely in my core area of expertise and interest, I have started a collaborative project with Prof. Stacy Patterson on privacy-preserving vertical federated learning using differential privacy and multi-party computation.

2.1 Compilation and Optimization for MPC

One direction builds new intra-procedural analyses and optimizations over MPC-IR. In addition to SIMD-vectorization, we develop divide-and-conquer, scheduling, protocol mixing and other optimizations. We extend classical program analysis techniques to the unique setting and constraints of MPC, but we also develop new MPC-specific cost models and optimizations, and aim to explore combinations and ordering of optimizations, much in the vein of the classical compiler. The key premise is that the linear structure of MPC-IR is highly amenable to program analysis, accurate cost modeling, and program synthesis and therefore these techniques can give rise to aggressive and provably optimal transformations.

Another direction builds a theoretical foundation for proving correctness of transformations over MPC-IR. The formalization of MPC-IR we have developed is a step forward; we aim to build an Abstract Interpretation-based framework for reasoning about transformations.

The third direction extends intra-procedural reasoning to the inter-procedural case. We enhance the source language with a notion of a secure component that executes under expensive secure computation protocols and a plaintext component, and enforce interaction and value flow between the two components with a flexible polymorphic (i.e., context-sensitive) type qualifier system. The type system is expressed within the pluggable types framework described in Sec. 1.3. A key goal is to integrate the type qualifier system (operating at a higher level) with backend-independent optimizations (at the lower level), towards an expressive and efficient programming system.
2.2 Principled Static Analysis of Python

I will also continue my work on principled and practical static analysis of Python. One direction develops classical analyses such as points-to analysis on top of PetPy and applies the analyses towards software tasks — type inference, specification inference and array shape analysis which have applications in reasoning about correctness of machine learning libraries.

Another direction expands on our “soundness” analysis. We aim to build a framework that estimates soundness of a static analysis for Python. Given an analysis, it will classify analysis results as sound and unsound based on assumptions the analysis makes in its interpretation of Python constructs. The analysis can make stronger assumptions and classify a larger subset of results as sound, and vice versa, it can make weaker assumptions resulting in a smaller subset of results deemed sound. In the most severe case the analysis will make no assumptions about the fall-through common interpretation; thus, results that interact with uninterpreted statements and expressions will be marked unsound. It is also important to allow for realistic assumptions. In more realistic cases the analysis will make certain assumptions — e.g., our soundness analysis for weakest preconditions does assume that the common fall-through implementation captures flow of references and reference mutation correctly.

This is a difficult problem. We aim to expand on principled analysis — in addition to specifying interpretation (which is mostly standard) we aim to build a framework for specifying assumptions. At the same time, the analysis will remain practical and will tackle real-world, widely-used Python software systems. The goal is to ground practical analysis around the PetPy syntax and allow for clear and concise statement of interpretation semantics and assumptions.

References


